Prototyp:

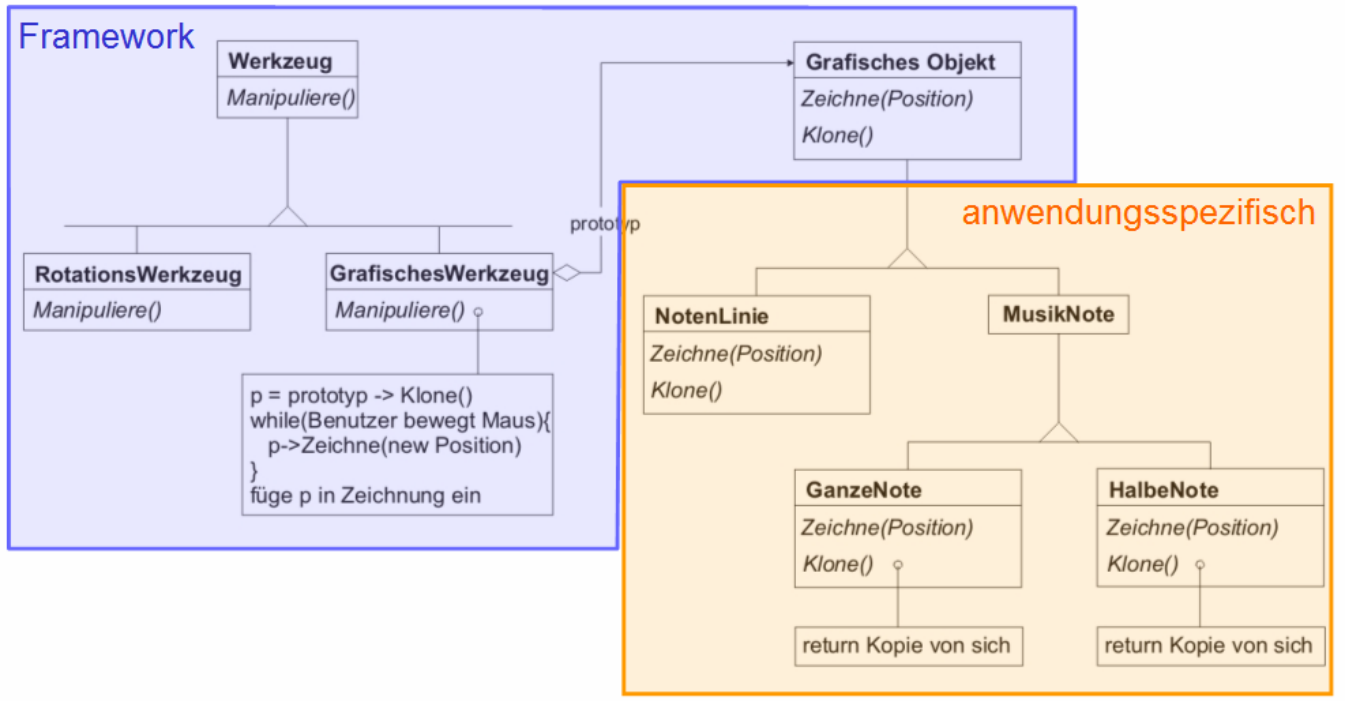
- Objektbasiertes Erzeugungsmuster (beschreibt Vorgehensweise zur Erzeugung von neuen Objekten)

Zweck:

- Erzeugung von neuen Objekten durch Kopieren (Klonen) eines prototypischen Exemplars
- Grundidee: „Erzeuge eine neue Instanz, indem du eine bestehende klonst und das entstandene Objekt an deine Bedürfnisse anpasst.“

Motivation:

- Illustration anhand eines Beispieles: gegeben sei ein allgemeines Framework für grafische Editoren
  → dieses ist anzupassen, um einen Editor für Musikpartituren zu erstellen: dieser beinhaltet u.a. das Hinzufügen von Noten, Pausen, Notenlinien,…
- Das Framework besitzt schon Werkzeuge zum Hinzufügen/Entfernen/Manipulieren von grafischen Objekten, welche dann in der Implementierung des Musikeditors z.B. Noten oder Notenlinien darstellen.
- *Klassendiagramm:*

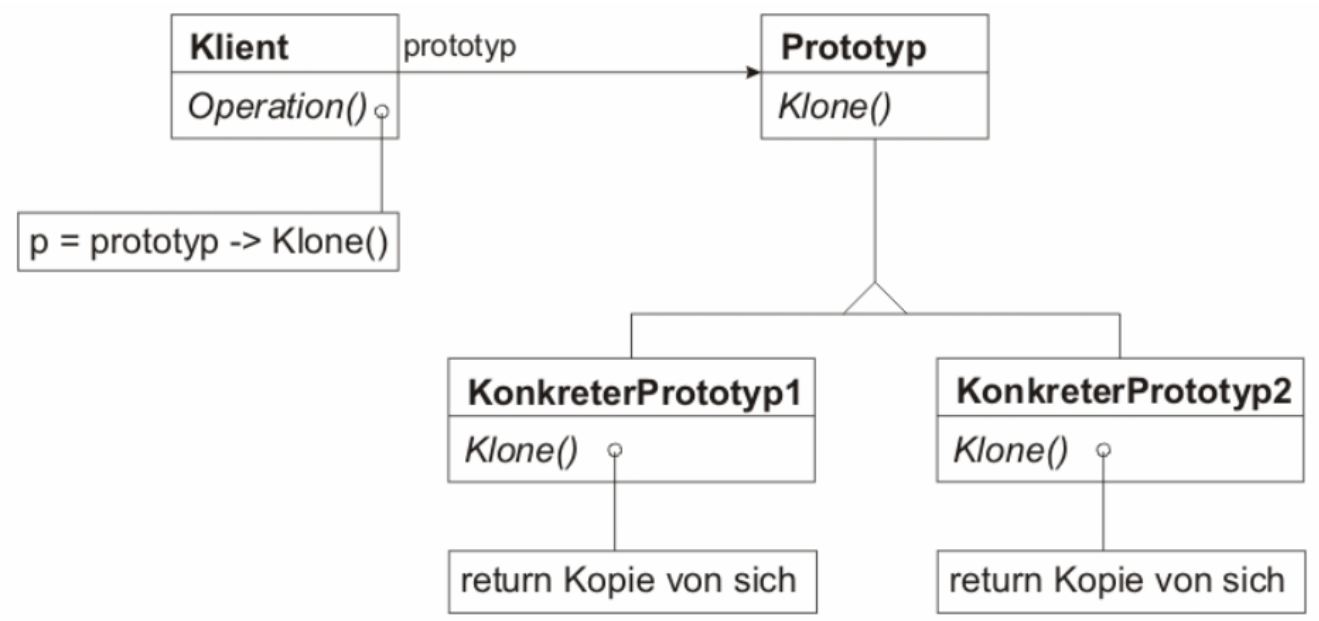


- *Problem:* „GrafischesWerkzeug“ gehört zum Framework, weiß also nichts von den anwendungsspezifischen Implementierungen von „GrafischesObjekt“, die es zu erzeugen hat (wie z.B. Noten oder Notenlinien)
- für jedes Musikobjekt könnte man nun eine eigene Unterklasse von „GrafischesWerkzeug“ bilden, dies würde aber zu vielen kleinen Unterklassen führen;
  Alternative zur Vererbungsstrategie: „GrafischesWerkzeug“ erzeugt neue grafische Objekte durch Kopieren/Klonen eines Exemplars einer Implementierung von „GrafischesObjekt“
  → dieses Exemplar heißt „Prototyp“
- „GrafischesWerkzeug“ wird also mit den verschiedenen Prototypen parametrisiert; im Musikeditor-Beispiel würde es z.B. verschiedene Instanzen von „GrafischesWerkzeug“ geben, welche alle mit einem anderen Prototypen parametrisiert sind;
  Verdeutlichung am Klassendiagramm: jeweils ein Objekt von „NotenLinie“, „GanzeNote“ und „HalbeNote“ bildet einen Prototyp; ein Werkzeug z.B. zur Erzeugung von ganzen Noten ist eine Instanz von „GrafischesWerkzeug“, welches mit „GanzeNote“ parametrisiert ist; dieses erzeugt eine neue ganze Note durch Klonen des Prototypen von „GanzeNote“ und Einfügen des geklonten Objekts in die Partitur

Anwendbarkeit:

- Prototypmuster kann vielfältig eingesetzt werden:
  - um ein System von der konkreten Repräsentation seiner Objekte unabhängig zu machen (wie im Framework-Beispiel „GrafischesWerkzeug“ unabhängig von den konkreten grafischen Objekten gemacht wurde),
  - um eine Hierarchie von Erzeuger-Unterklassen zu vermeiden (die im Beispiel von „GrafischesWerkzeug“ abgeleitet sein würden),
  - um erst zur Laufzeit zu spezifizieren, von welchen Klassen Objekte erzeugt werden sollen,
  - um neue Objekte in das System aufzunehmen (z.B. ein bearbeitetes grafisches Objekt in eine Liste von Standardobjekten),
  - wenn sich Objekte einer Klasse ähneln → in diesem Fall kann es einfacher und „billiger“ sein Prototypen einzurichten und sie zu klonen anstatt Konstuktoren aufzurufen und die Objekte jedes Mal neu anzupassen.

Struktur:



Teilnehmer:

- *Klient:* erzeugt neue Objekte durch Klonen seines Prototypen und modifiziert diese anschließend
- *Prototyp:* deklariert Schnittstelle für das Klonen
- *KonkreterPrototyp:* implementiert Schnittstellenmethode „Klone“

Interaktionen:

- Beim Erzeugen neuer Objekte befiehlt der Klient einem Prototypen sich selbst zu klonen, wobei der Prototyp und auch die Inhalte aller seiner Variablen kopiert werden, womit auch komplexe Objektstrukturen kopiert werden können.

Konsequenzen:

- Konkrete Produktklassen werden vor dem Klienten verborgen, der damit nur die Schnittstelle sieht
- Klient kann wie bei unserem Framework-Beispiel ohne Modifikation mit anwendungsspezifischen Klassen arbeiten
- Weitere Möglichkeiten:
  1. *Hinzufügen/Entfernen von Prototypen zur Laufzeit:*
     Instanzen einer neuen Produktklasse können zur Laufzeit in das System eingebunden werden, indem man deren Prototyp beim Klienten registriert
  2. *Definieren neuer Objekte durch Variation von Werten:*
     Erzeugung neuer Arten von Objekten zur Laufzeit, indem Objekte in ihren Variablen angepasst und beim Klienten registriert werden → damit muss zum Erhöhen der Funktionalität keine neuen Klassen programmieren, wodurch auch die Anzahl benötigter Klassen sinkt
  3. *Definieren neuer Objekte durch Variation der Struktur:*
     Beispiel Vektorgrafik → oft will man eine komplexe erzeugte Grafik als ein Objekt ansehen und in mehreren Zeichnungen wieder verwenden; mit dem Prototypmuster kann man dies realisieren, indem man die Grafik als neuen Prototyp registriert und in die Palette zur Verfügung stehender Objekte aufnimmt
  4. *Dynamisches Konfigurieren einer Anwendung mit „Klassen“:*
     wenn Objekte erst zur Laufzeit geladen werden sollen, dann kann man diese als Prototypen registrieren und statt einem Konstruktor nur die Methode Klone() aufrufen
- *Problem:* beim Prototypmuster muss jede Unterklasse des Prototyps die Operation „Klone“ implementieren, was zu einer schwierigen Aufgabe werden kann und teilweise umständlich zu realisieren ist (dazu kommen wir noch bei der Problembetrachtung zum „Kopieren“)

Implementierung:

- Verschiedene Aspekte zu beachten:
  1. *Verwendung eines Prototypenverwalter:*
     Ist besonders nützlich, wenn die Prototypanzahl nicht von vornherein festgelegt bzw. recht hoch ist. Hier verwalten Klienten die Prototypen nicht selbst, sondern speichern sie in einer Registratur ab, dem Prototypenverwalter. Dieser assoziiert Prototypobjekte mit entsprechenden Namen (Codes), über die Klienten auf die Prototypen zugreifen können.
     Beispielcode zum Erzeugen einer neuen Halbnote mit Zugriff auf den Prototypmanager:

```
Object *p = protoVerwalter(HALBNOTE)->Klone();
p->init(x,y,z);
//HALBNOTE ist der Code/Schlüssel für eine Halbnote
//zunächst erfolgt das Klonen, dann wird das neue Objekt initialisiert
```

  2. *Implementierung der Klone-Operation:*
     Die korrekte Implementierung kann schwierig sein. Allgemein gibt es 2 Möglichkeiten, auf die weiter unten näher eingegangen wird: die „flache Kopie“ und die „tiefe Kopie“ .
     Manche Klienten erfordern die Initialisierung von geklonten Objekten mit Werten ihrer Wahl → im Normalfall kann dies NICHT direkt über die Klone-Operation erfolgen, weil die Anzahl der zu initialisierenden Werte je nach Prototyp verschieden sein kann. Somit muss, wenn nicht bereits vorhanden, eine „Initialisiere“-Operation eingeführt werden, die je nach Prototyp bestimmte Parameter entgegen nimmt und die Klassenvariablen entsprechend setzt.

Beispielcode (C++):
Implementation der Klasse „GrafischesWerkzeug“ (unter Verwendung eines gegebenen Typs „Bitmap“):

```
class GrafischesWerkzeug : public Werkzeug{
      public:
            GrafischesWerkzeug(GrafischesObjekt*);
            virtual GrafischesObjekt* ErzeugeGrafischesObjekt() const;
      private:
            GrafischesObjekt* _prototyp_graf_objekt;
};

GrafischesWerkzeug::GrafischesWerkzeug(GrafischesObjekt* graf_objekt){
      _prototyp_graf_objekt=graf_objekt;
}

GrafischesObjekt* GrafischesWerkzeug::ErzeugeGrafischesObjekt(Bitmap*) const{
      GrafischesObjekt* klon = _prototyp_graf_objekt->Klone();
      klon->Initialisiere(Bitmap);
      return klon;
}
```

Eine vereinfachte Implementierung der Klasse „HalbeNote“ könnte wie folgt aussehen, vorausgesetzt sie ist eine Unterklasse von „MusikNote“, deren Basisklasse „GrafischesObjekt“ ist (siehe Klassen-Diagramm):

```
class HalbeNote : public MusikNote{
      public:
            HalbeNote();
            HalbeNote(const HalbeNote&);
            virtual GrafischesObjekt* Klone() const;
            virtual void Initialisiere(Bitmap*);
      private:
            Bitmap* _bild;
};

HalbeNote::HalbeNote(const HalbeNote& andereHalbnote){
      _bild=andereHalbnote._bild; //flache Kopie
}

void HalbeNote::Initialisiere(Bitmap* bild){
      _bild=bild;
}

GrafischesObjekt* HalbeNote::Klone() const{
      return new HalbeNote(*this);
}
```

Ein grafisches Werkzeug zur Erzeugung von Halbnoten könnte nun z.B. wie folgt definiert werden:

```
GrafischesWerkzeug WerkzeugFuerHalbnoten(new HalbeNote);
```

Eine neue halbe Note würde dann wie folgt erzeugt werden:

```
Bitmap* bild = new Bitmap(„halbenote.bmp“); //Bild zur Darstellung einer Halbnote
HalbeNote* neueHalbnote = WerkzeugFuerHalbnoten.ErzeugeGrafischesObjekt(bild);
```

Obwohl Klone() hier einen Zeiger auf ein Objekt vom Typ „GrafischesObjekt“ zurückgibt, wird bei der Implementation ein Zeiger vom Typ „HalbeNote“ zurückgegeben. So implementiert muss der Klient nichts von der konkreten Unterklasse wissen, welche er klont und muss keinen Downcast zum erwünschten Typ ausführen. Im Beispiel wäre das Framework somit losgelöst von der konkreten Implementierung des Musik-Editors.

Gesamtüberblick:

- Ein Prototyp ist ein Objekt einer Klasse, welches geklont werden kann um neue Objekte zu erzeugen.
- Grundidee: „Erzeuge eine neue Instanz, indem du eine bestehende klonst und das entstandene Objekt an deine Bedürfnisse anpasst.“
- Einsatz von Prototypen wenn man komplizierte Objekte nicht jedes Mal neu per Hand erstellen will oder z.B. zur Laufzeit neue „Objektklassen“ angelegt werden sollen.
  Ebenso kann damit eine aufwendige Erzeuger-Klassenhierarchie vermieden werden.

Kopieren:

- Um praxisnah zu bleiben und ein wenig Java zu vermitteln, möchten wir uns an dieser Stelle ein wenig von den Prototypen lösen und auf die Problematik des Kopierens eingehen. (Dies spielt natürlich auch bei den Prototypen eine große Rolle, da in der Klone()-Methode genau das realisiert werden muss.)

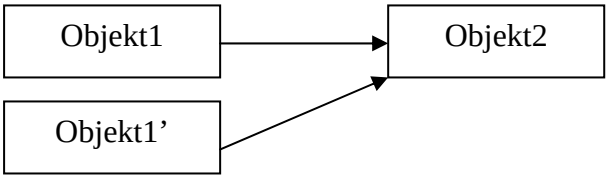
Beispiel der 2. Übung:

- Dazu wollen wir das Beispiel aus der letzten Übung heranziehen. Sieht man sich die Klasse „Display“ an, so wird darin ein Objekt „c“ mit „Stepper c = new Counter();“ initialisiert. Was geschieht, wenn wir ein Objekt „d“ wie folgt einführen: „Stepper d = c;“? Wird damit eine Kopie von „c“ erstellt?
  Dem ist nicht so: vielmehr ist „d“ nur ein Verweis auf „c“, d.h. ändert man den internen Zustand von „d“ z.B. durch „d.step()“, so wird auf den Speicher von „c“ zugegriffen und dieses Objekt ebenfalls verändert.
- Wie lässt sich nun eine Kopie von „c“ erstellen? Dafür gibt es 2 Möglichkeiten:

1. <u>„flache Kopie“:</u>

Beim Kopieren einer Klassen-Instanz werden bei internen Variablen, die Referenzen auf Objekte darstellen, nur die Referenzen kopiert anstatt des kompletten referenzierten Objekts. Dadurch werden Objekte mit Referenzen von Original und Kopie gemeinsam genutzt, was gewollt sein kann, meist aber unbeabsichtigt ist. Im Beispiel der Prototypen würde dies dazu führen, dass die Änderung eines Klons eine Änderung des Prototyp-Objekts und aller weiteren Klone zur Folge hätte, was sicher nicht beabsichtigt ist.

Grafische Darstellung:



In Java:
Für das flache Kopieren stellt Java schon einen Mechanismus bereit, der aus dem Interface „java.lang.Cloneable“ und aus der Methode „clone()“ der Klasse „Object“ besteht. Dadurch, dass „clone()“ in „Object“ definiert ist, steht es in allen Klassen und z.B. auch für Arrays zur Verfügung.

Will eine Klasse mit „clone()“ bitweise Kopien von ihren Instanzen erstellen, muss die betreffende Klasse das Interface „Cloneable“ implementieren.
Falls „clone()“ auf ein Objekt angewendet wird, dessen Klasse dieses Interface nicht implementiert, wird eine „CloneNotSupportedException“ geworfen, welche beim Aufruf von „Object.clone()" immer abgefangen werden muss.

Beispiel anhand der Klasse Counter:

```
public class Counter implements Stepper,Cloneable {
      //...

      public Object clone(){
            Object klon = null;
            try{
                  klon = super.clone();
            } //Exception MUSS abgefangen werden
            catch(CloneNotSupportedException e) {}
            return klon;
      }
}
```

Aufruf in der Klasse Display:

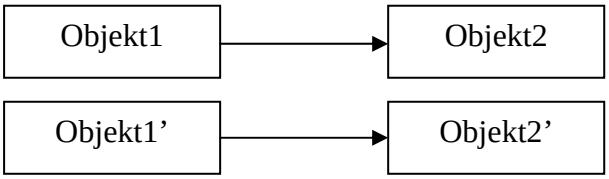
```
final Stepper d = (Counter)c.clone(); //flaches Kopieren
```

Dieses Java-Konzept erweist sich als sehr sinnvoll. In C++ müsste man z.B. einen Copy-Konstruktor definieren, welcher eine neue Instanz der Klasse erzeugt und die Variablen dieser Instanz einzeln initialisiert. Hier kann man sich mit Java doch einiges an Arbeit sparen.

2. „tiefe Kopie“:

Zusätzlich zur flachen Kopie werden rekursiv Kopien von den referenzierten Objekten erzeugt. Um Objekte unabhängig voneinander zu machen und damit Konflikte zu vermeiden, werden also nicht nur die Referenzvariablen in der zu kopierenden Instanz geklont, sondern auch die Objekte, welche hinter diesen Referenzen stehen.

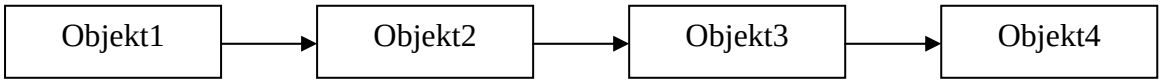
Grafische Darstellung:



In Java:

Standardmäßig bietet Java keine direkte Möglichkeit zum tiefen Kopieren, dies muss vom Entwickler eigenhändig realisiert werden, was sich als schwierig herausstellen kann, wenn Objekte rekursiv über mehrere Ebenen kopiert werden müssen.

Beispiel:

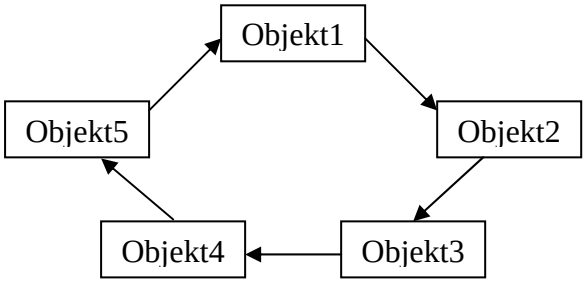


Hier müsste bei einer tiefen Kopie von Objekt1 jedes Objekt, auf welches in einem der Objekte verwiesen wird, rekursiv kopiert werden.

Problem der zirkulären Referenzen:

Hier ist das Erstellen einer tiefen Kopie besonders trickreich.

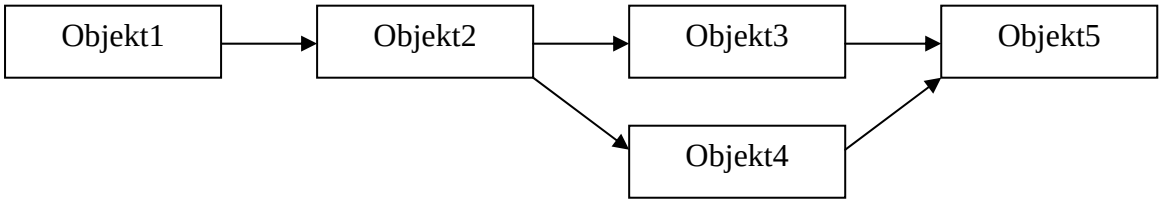
Zur Veranschaulichung sei folgende Grafik gegeben:



Wie würde man jetzt z.B. eine tiefe Kopie von Objekt1 erstellen? Würde man rekursiv Kopien aller Objekte erzeugen, so würde bei Objekt5 wieder rekursiv Objekt1 kopiert werden, man würde sich quasi „im Kreis drehen“ und es würden Kopien erstellt werden, bis der Speicher voll ist.

Bei solchen „Ringlisten“ und in Fällen, wo Objekte gegenseitig aufeinander zeigen, kann nicht einfach rekursiv kopiert werden. Vielmehr liegt die Verantwortung für das Verhalten beim Programmierer: er muss z.B. durch Programmierung eines „Überwachers“ sicherstellen, dass erkannt wird, welche Objekte schon kopiert wurden und die Referenzen auf schon kopierte Objekte korrekt setzen.

Ähnliche Fälle lassen sich leicht konstruieren, z.B.:



oder auch: